



MINISTÉRIO DA CIÊNCIA, TECNOLOGIA, INOVAÇÕES E COMUNICAÇÕES  
**INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS**

# Algoritmos e Estruturas de Dados

## Lista de exercícios 1

Felipe Menino Carlos

Este documento tem por objetivo apresentar considerações sobre alguns dos vários tópicos abordados durante o desenvolvimento da 1ª lista de exercícios de Algoritmos e Estruturas de Dados.

### 1 Exercício 2

Para o exercício 2 apresentado na lista, que tinha como objetivo a expansão das operações disponíveis da lista encadeada implementada durante as aulas, pede-se a classificação das complexidades de cada uma das operações implementadas. Abaixo são listados cada uma das operações implementadas e suas respectivas complexidades.

**Tabela 1:** Complexidade das operações implementadas na lista ligada

Operação	Complexidade	Descrição
front()	$O(1)$	Retorna o primeiro elemento da lista
back()	$O(1)$	Retorna o último elemento da lista
push_front(v)	$O(1)$	Insere o valor v na cabeça da lista
push_back(v)	$O(1)$	Insere o valor v no final da lista
pop_front()	$O(1)$	Remove o primeiro elemento da lista
pop_back()	$O(1)$	Remove o último elemento da lista
splice(L2)	$O(2N)$	Funde os elementos da lista L2 ao final da lista operada
reverse()	$O(N)$	Reverte os elementos da lista
merge(L2)	$O(N^2)$	Junta duas listas ordenadas em uma só, também ordenada

Vale citar que, a complexidade da operação *Merge(L2)* tem o formato apresentado por conta do algoritmo selecionado, sendo este o Tree Sort. McLuckie (1986) [1] apresenta a descrição do algoritmo.

## 2 Exercício 4

O exercício 4 pede a implementação de um *deque* para permitir inserções e remoções de elementos nas duas extremidades em tempo constante  $O(1)$ , garantindo também o acesso ao  $i$ -ésimo elemento em tempo constante  $O(1)$ . Além disto, inserções que não ocorram nas extremidades poderão ser feitas em tempo  $O(N)$ .

Inicialmente como forma de solução para este exercício foi adotado uma abordagem que faz a utilização de listas ligadas, controlando inserções e remoções em tempo constante, porém, com esta forma de implementação o acesso ao  $i$ -ésimo elemento não é em tempo constante  $O(1)$ , já que, nesta abordagem é necessário percorrer cada um dos elementos da lista ligada até chegar no selecionado, como apresentado no pseudocódigo abaixo.

---

### Algorithm 1 Algoritmo de busca em lista ligada

---

```

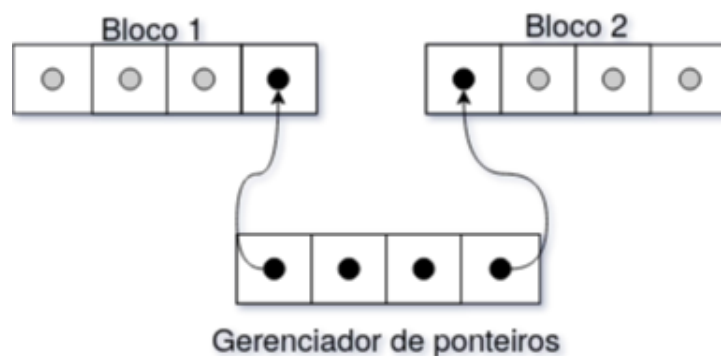
1: function BUSCAEMLISTALIGADA(A, el)
2:    $item \leftarrow A.next$ 
3:   while item != A.end and item.data != el do
     item = item.next
4:   end while
5:   return item
6: end function

```

---

Com isto, foi necessário a identificação de uma outra abordagem para que os requisitos base da estrutura de dados proposta no exercício fossem atendidos. Inicialmente fez-se uma busca pela estrutura de dados equivalente na Standard Template Library (STL) do C++, esta que como apresentado na documentação<sup>1</sup> garante o acesso ao  $i$ -ésimo elemento em tempo constante  $O(1)$ . Musser *et al* (2001) [2] especificam que a STL faz a garantia do tempo constante através de uma implementação que trabalha com o gerenciamento de ponteiros e múltiplos blocos de tamanho fixo. Uma representação para tal implementação pode ser vista na Figura 1.

**Figura 1:** Representação da forma de implementação do Deque na STL



Para este exercício uma abordagem mais simples foi adotada, onde através de um *array* estático de armazenamento contíguo foi utilizado. Este *array* possui uma representação

---

<sup>1</sup><https://en.cppreference.com/w/cpp/container/deque>

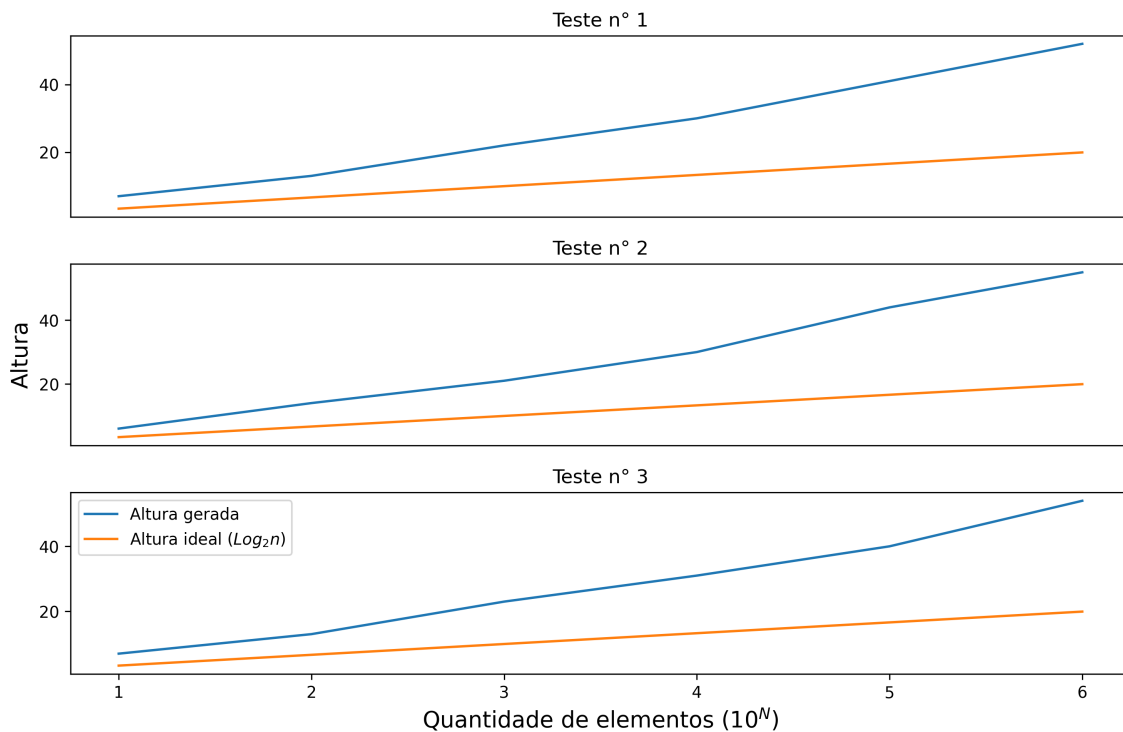
circular dentro da estrutura de dados implementada, sabendo que, com a utilização de dois ponteiros de memória, um representando o início do *deque* e outro representando o fim, todo o comportamento padrão de inserção, remoção e busca foram implementados.

A estrutura de dados implementada neste exercício pode apresentar problemas de usabilidade por conta da necessidade de definição da quantidade de elementos a serem armazenados em tempo de compilação e por não possibilitar o aumento da capacidade em tempo de execução. Por outro lado, garante velocidade  $O(1)$  em boa parte das operações implementadas.

### 3 Exercício 9

A altura das árvores binárias é definida como  $\log_2 n$ , porém, como apresentado por Sedgwick e Wayne (2011) [3] tal característica só é garantida quando a árvore é balanceada, ou seja, quando a altura das duas subárvores de todo nó nunca difere mais que um elemento. Com isto, ao analisar os resultados obtidos com o teste do exercício nove, apresentados na Figura 2, é possível perceber tal comportamento, isso já que, a implementação de árvore binária realizada neste trabalho não faz nenhum tipo de balanceamento.

**Figura 2:** Testes com a altura de árvore binária criada aleatoriamente



## Referências

- [1] MCLUCKIE, K. *Sorting routines for microcomputers*. Macmillan, Basingstoke, 1986.
- [2] MUSSER, D., DERGE, G., AND SAINI, A. *STL tutorial and reference guide : C++ programming with the standard template library*. Addison-Wesley, Boston, 2001.
- [3] SEDGEWICK, R., AND WAYNE, R. *Algorithms*. Addison-Wesley, Upper Saddle River, NJ, 2011.